# Calico Graphics

### From IPRE Wiki

This page describes Calico Graphics, a 2D graphics library for creating art, games, and animations in any of the Calico languages.

See also Calico Differences which lists the differences between CPython Myro and IDLE.

You will find a set of examples at http://svn.cs.brynmawr.edu/viewvc/Calico/trunk/examples/ in various languages.

## Contents


- 1 Importing Graphics
  - 1.1 Calico Python
- 2 Windows
  - 2.1 Windows Methods
  - 2.2 Window Modes
  - 2.3 Mouse
  - 2.4 Key Presses
- 3 Shapes
  - 3.1 Shape Properties
  - 3.2 Shape Methods
  - 3.3 Gradient
  - 3.4 Arrow
  - 3.5 Dot
  - 3.6 Curve
  - 3.7 Text
  - 3.8 Line
  - 3.9 Picture
    - 3.9.1 Pixel
    - 3.9.2 Colors
  - 3.10 Rectangle
  - 3.11 RoundedRectangle
  - 3.12 Polygon
  - 3.13 Circle
  - 3.14 Oval
  - 3.15 Pie
  - 3.16 Arc
  - 3.17 GUI Widgets
    - 3.17.1 Button
    - 3.17.2 HSlider
  - 3.18 Groups of Shapes
    - 3.18.1 Frame
    - 3.18.2 Graphs
      - 3.18.2.1 Scheme Example
    - 3.18.3 Group
- 4 Window Features
  - 4.1 modes
  - 4.2 Miscellaneous
  - 4.3 Physics
- 5 Low Level Graphics

- 6 Plot


# Importing Graphics

The Calico Graphics library allows for the creation of graphical objects in a window. The library Myro contains just a few of the Graphics functions:

- Window(...)
- Color(...)
- getColorNames()
- getPixels(...)
- Vector(...)
- copyPicture(...)
- getWindow()
- getMouse()
- getMouseNow()
- getMouseState()
- getKeyState()
- getKeyPressed()
- makePicture(...) (for Picture())

Graphics is fully available for Python, Ruby, F#, and Scheme.

Python:

```
python> import Graphics
```

Ruby:

```
ruby> require "Graphics"
```

Scheme:

```
scheme> (using "Graphics")
```

F#:

```
fsharp> #r "Graphics.dll";;
```

## Calico Python

The rest of this page describes the Graphics library using Python syntax.

For example, you can use:

```
from Myro import *
```

to include the Graphics functions and classes. Of course, that will also import many other functions not related to Graphics as well.

To just load all of the defined functions from the Graphics library in Calico, one could:

```
from Graphics import *
```

This will make all of the functions and classes available directly:

```
circle = Circle((100,200), 20)
```

Or, you could also:

```
import Graphics
```

However, then you must always use the "Graphics." prefix on functions and classes:

```
circle = Graphics.Circle((100,200), 20)
```

You can also just import the items you want, like so:

```
from Graphics import Circle
```

The following code assumes that you have imported all of the functions and classes from either Myro or Graphics.

# Windows

All shapes are drawn on a Window. You must create a window before you can draw onto it:

```
win = Window()
win = Window("Title of Window")
win = Window(width, height)
win = Window("My Title", width, height)
```

Window can take three optional arguments: title, width, and height. The defaults are "Calico Graphics" and 300 width by 300 height. For example:

```
win2 = Window("My Second Window")           # creates a window with title "My Second Window" that is 300 x 300
win3 = Window("My Third Window", 400)       # creates a window with title "My Third Window" that is 400 x 300
win4 = Window("My Fourth Window", 400, 600) # creates a window with title "My Fourth Window" that is 400 x 600
win5 = Window(500, 500) #creates an anonymous window that is 500 x 500
```

You can always get the last window *created* using the getWindow() function:

```
Window("New Window")
win = getWindow()
```

Note that Graphics will use reuse windows based on the title. For example:

```
win1 = Window()
win2 = Window()
```

will only create one window, and both win1 and win2 will refer to it. To create two windows, use different titles:

```
win1 = Window("Window 1")
win2 = Window("Window 2")
```

# Windows Methods

```
window.addScrollbars(width, height)
```

Adds scrollbars on a window and expand the underlying canvas to width x height.

# Window Modes

Calico Windows operate in one of four modes: "auto", "manual", "bitmap", "bitmapmanual", or "physics". If in "auto", "bitmap", or "physics" mode, then changes to the window are made as soon as possible so that you can see their effect. However, if too many updates are made, then you will only see the changes occur every win.updateInterval seconds. By default, win.updateInterval is .1 seconds (100 milliseconds).

The "bitmap" and "bitmapmanual" modes will not render shapes as shapes, which are redrawn each time them window needs to be re-rendered, but draws them once to a bitmap area associated with the window. This makes redrawing very fast, but at the cost of being unable to treat shapes as objects. For example, you would not be able to "move" a shape if it were drawn in bitmap mode.

The "manual" and "bitmapmanual" modes are designed so that you can control how fast the window updates, and makes sure that all of the objects update together. You must use the win.step() (or win.step(SECONDS)) method when you have the win.mode set to "manual" or "bitmapmanual". The window will not update faster than SECONDS. By default, SECONDS is 0, so it will update as soon as you call step().

Examples:

If you want the system to update automatically (the default), use:

```
win.mode = "auto"               # this line is not needed, as this is the default
win.updateInterval = .1         # this line is optional
```

That will update the window no more than every .1 seconds. This helps make your computer more efficient, so that it only makes updates every so often. You wouldn't want the computer to update the screen for every pixel change or else your graphics would slow to a crawl. If you want to force an update in this mode, use:

```
win.update()
```

If you want to control exactly when a window updates, then use:

```
win.mode = "manual"
...
win.step(.1)                    # or win.step()
```

This will force the update when you call step(.1), but will make sure that it doesn't update faster than every 100 ms. That number allows you to speed up and slow down animations.

Setting mode to "auto" is good for interactive use, but "manual" and using step() is good for making animations.

```
win.mode = "bitmap"
Line((0, 0), (100, 100)).draw(win)
```

Here the Line is created, and drawn to the bitmap of the window. There is no use saving the Line object. You can switch back and fourth between bitmap and manual/auto. That will allow you to have some objects, and some drawings.

The "physics" mode is discussed below.

# Mouse

There are three main functions for dealing with mouse events:

- getMouse() - waits until user clicks and returns (x, y) of location in window
- getMouseNow() - gets (x, y) of mouse location, relative to window
- getMouseState() - gets current mouse state ("up" or "down")

You can also attach functions to mouse events so that the function will define what to do.

- onMouseMovement(function)
- onMouseDown(function)
- onMouseUp(function)

You can attach as many functions as you like. Each function takes the window and the event:

```
def handleMouseUp(obj, event):
    print("Up")

def handleMouseDown(obj, event):
    print("Down")

def handleMouseMovement(obj, event):
    print("Movement")

win = Window()
onMouseMovement(handleMouseMovement)
onMouseUp(handleMouseUp)
onMouseDown(handleMouseDown)
```

All of these mouse functions are also methods of the Window class. So if you have multiple windows, you can call the function directly:

```
win = Window()
win.getMouse()
win.getMouseNow()
win.getMouseState()

win.onMouseMovement(handleMouseMovement)
win.onMouseUp(handleMouseUp)
win.onMouseDown(handleMouseDown)
```

The obj argument is (in these cases) a Graphics.WindowClass instance, and the event is an Event object:

- event.x - x component of event
- event.y - y component of event
- event.time - time of event, in global seconds
- event.type - returns name of event type: "mouse-release", "mouse-press", "mouse-motion", "key-press", "key-release"
- event.key - returns string key name from a key event

And for key events:

- event.type - "mouse-motion", "mouse-press", "mouse-release", "key-press", "key-release"
- event.key - key name of press or release (eg, "Return", "A", "Tab", etc.)

Other events have various obj and event properties:

- event.type = "click"; obj is Button - for Button widget
- event.type = "change-value", event.value is 0 to 101; obj is Slider - for HSlider and VSlider

See also the joystick(), getGamePad() and getGamepadNow() functions of the Calico Myro library.

## Key Presses

There are two main functions for dealing with keyboard events:

- getKeyPressed() - gets a string of the last key pressed
- getKeyState() - gets the state of the keyboard (a key "up" or a key "down")

As with the mouse, you can also attach functions to keyboard activity:

- onKeyPress(funtion)
- onKeyRelease(function)

These functions, like the mouse event handlers, are passed the window and the event:

```
def handleKeyPress(win, event):
    print("Down")

def handleKeyRelease(win, event):
    print("Up")

win = Window()
onKeyPress(handleKeyPress)
onKeyRelease(handleKeyRelease)
```

The KeyPress event will be called continuously while the key is down.

The win argument is a Graphics.WindowClass instance, and the event is a Event object (see above).

# Shapes

You can create and draw a variety of shapes in a window. Each shape results in a representation on a window when drawn. However, it is also an object which can be moved around, changed in various ways, and even undrawn.

Shapes require Points to define their boundaries or center. Wherever a Point is required, you can optionally use a list or tuple.

## Shape Properties

All shapes have:

- fill - color of area of shape
- outline - color of border of shape
- color - shortcut for setting both fill and outline (need to set to a color first)
- gradient - (as opposed to fill) - a Gradient() (see below)
- border - thickness of border of shape
- pen - leaves trail if penDown()
- body - for Physics simulations
- bounce - 1.0 is 100% bounce; 0.0 is no bounce
- friction - amount of friction
- density - set this before you draw it
- mass - set this after you draw it (with win.mode set to "physics")
- wrap - set to True to keep shape on the screen
- bodyType - either "static" or "dynamic"
- center
    - center.x
    - center.y
- points - points that make up a shape (if needed)
- tag - string for labeling item

- window - the (last) window a shape is drawn in
- rotation
- scaleFactor

## Shape Methods

All shapes have:

- shape.draw(window) - put shape on window's list of items to draw
- shape1.draw(shape2) - draw shape1 into shape2's reference frame
- shape.move(dx, dy) - move by delta dx, delta dy
- shape.moveTo(x, y) - move to x, y
- shape.rotate(degrees) - rotate by degrees (positive is counter-clockwise)
- shape.rotateTo(degrees) - rotate to degrees (0 is to the right, moving positive clockwise)
- shape.scale(factor) - scale by a percent (.1 is 10% of original; 1.1 is 10% larger)
- shape.scaleTo(factor) - set scale to factor (1 is 100% of original)
- shape.forward(distance) - move shape in its zero rotate direction

- shape.penDown() - put the shapes pen down to allow trace when it moves
- shape.penUp() - put the pen up and stop trace. Also returns a list of points

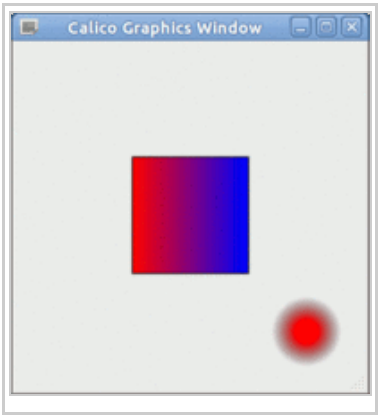
- shape.getScreenPoint(p) - given a point relative to the center of a shape, returns a point in global screen coordinates
- shape.getP1() - returns first point of a shape in screen coordinates
- shape.getP2() - returns second point of a shape in screen coordinates
- shape.getX() - get the x component of the center of a shape
- shape.getY() - get the y component of the center of a shape
- shape.setX(x) - set the x component of the center of a shape
- shape.setY(y) - set the y component of the center of a shape

## Gradient

A Gradient is composed of two colors and a transition between them.

- Gradient("linear", point1, color1, point2, color2)
- Gradient("radial", point1, radius1, color1, point2, radius2, color2)

```
from Graphics import *

win = Window()

sq = Rectangle((100, 100), (200, 200))
sq.gradient = Gradient("linear", (-50, 0), Color("red"), (50, 0), Color("blue"))
sq.draw(win)

c = Circle((250, 250), 50)
c.gradient = Gradient("radial", (0, 0), 10, Color("red"),
                      (0, 0), 30, Color(0, 0, 0, 0))
c.outline = None
c.draw(win)
```

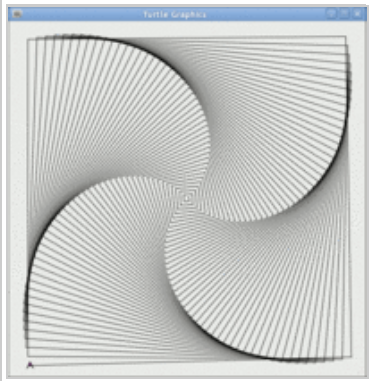



## Arrow

- Arrow((x, y), degrees) - create an arrow at (x,y) facing degrees (0 is to right)
- Arrow(Point(x, y), degrees) - create an arrow at (x,y) facing degrees (0 is to right)

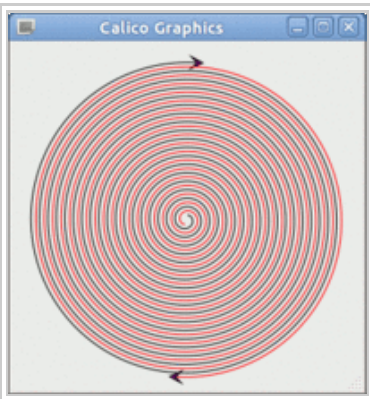
An arrow is a triangular pointer. Often used as the shape for creating "turtle graphics".

```
# Turtle Graphics Example for Calico Python
# After http://en.wikipedia.org/wiki/File:Turtle-Graphics_Polyspiral.svg
# Doug Blank <dblank@cs.brynmawr.edu>

from Graphics import *

size = 600
win = Window("Turtle Graphics", size, size)
turtle = Arrow((size/2, size/2), 0)
turtle.draw(win)
turtle.penDown()

def f(dist, angle, incr, segs):
  for i in range(segs):
    turtle.forward(dist * (size * .35))
    turtle.rotate(-angle)
    dist += incr

f(.01, 89.5, .01, 184)
```



Turtle Graphics

```
from Graphics import *
win = Window()
def spiral(x, y, d, loops, color):
   arrow = Arrow((150, 150))
   arrow.pen.color = color
   arrow.draw(win)
   arrow.moveTo(x, y)
   arrow.rotateTo(d)
   arrow.penDown()
   i = 0.0
   while i < .180 * loops:
       arrow.rotate(-2)
       arrow.forward(.1 + i)
       i = i + .001

spiral(150, 150, 0, 18, Color("black"))
spiral(148, 158, 180, 18, Color("red"))
```



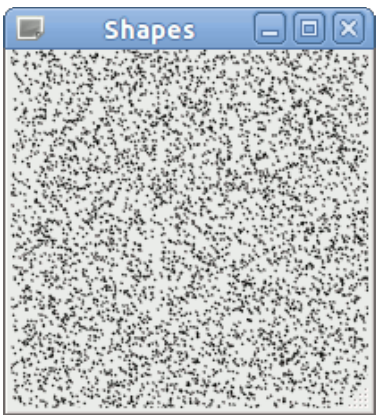

Spirals

## Dot

You cannot draw a Point; use Dot instead.

- Dot((x,y)) - draw a point at (x,y)
- Dot(Point(x,y)) - draw a point at (x,y)

```
from Graphics import *
import random
win = Window("Shapes", 200, 200)
for i in range(5000):
    shape = Dot(random.random() * 200, random.random() * 200)
    shape.draw(win)
```

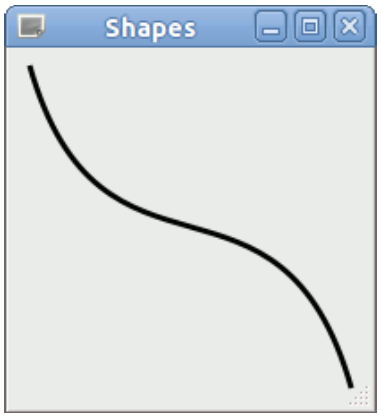



## Curve

P1 and P4 are the end points; P2 and P3 are control points.

- Curve((x1, y1), (x2, y2), (x3, y3), (x4, y4))
- Curve(Point(x1, y1), Point(x2, y2), Point(x3, y3), Point(x4, y4))

```
from Graphics import *
win = Window("Shapes", 200, 200)
curve = Curve((10, 10), (50, 150), (150, 50), (190, 190))
curve.border = 3
curve.draw(win)
```




## Text

Place text on the window.

- Text((x, y), text)
- Text(Point(x, y), text)

- text.fontFace - typeface of font
- text.fontWeight - bold?
- text.fontSlant - italics?
- text.fontSize - size of font
- text.xJustification - "center" (or "left" or "right")
- text.yJustification - "center" (or "top" or "bottom")
- text.width - (readonly) width of text in pixels
- text.height - (readonly) height of text in pixels

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = Text((100, 100), "Hello, World!")
shape.fill = Color("blue")
shape.rotate(45)
shape.draw(win)
```


## Line

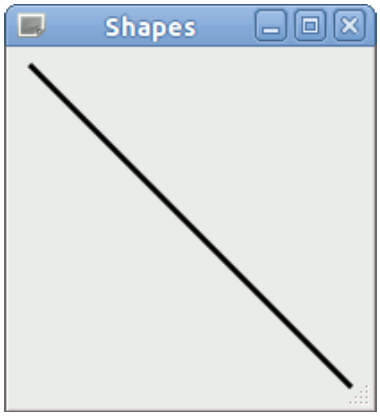
Draw a line.

- Line((x1, y1), (x2, y2))
- Line(Point(x1, y1), Point(x2, y2))

or

- line = Line()
- line.append(Point(x,y))
- line.set_points()

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = Line((10, 10), (190, 190))
shape.fill = Color("blue")
shape.border = 3
shape.draw(win)
```




# Picture

Display a picture.

- Picture(filename) - filename is a string including path
- Picture(window) - make a picture of the contents of a window
- Picture(URL) - get a picture from the web
- Picture(width, height) - make a blank picture width by height
- Picture(width, height, color) - make a blank picture width by height of a particular color
- Picture(picture) - makes a copy

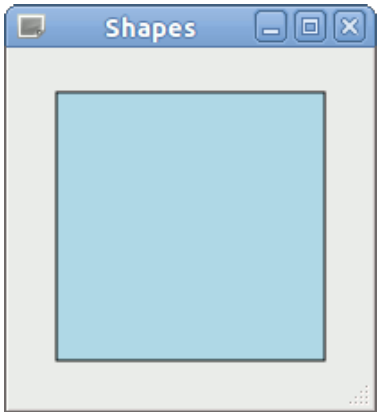
Additional functions:

- savePicture(picture, filename) - saves image in JPG or PNG format
- savePicture([picture, ...], filename.gif) - saves an animated GIF
- savePicture([picture, ...], filename.gif, delay) - saves an animated GIF
- savePicture([picture, ...], filename.gif, delay, loop?) - saves an animated GIF
- copyPicture(picture) - returns a copy of a picture
- Can also use makePicture(...) for all versions of Picture(...)
- getRegion(center_point, width, height, degrees) - get a region of an image
- setRegion(center_point, width, height, degrees, color) - set a region of an image to color

Properties:

- alpha - change the alpha of all pixels
- width - width of image
- height - height of image

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = Picture(190, 190, Color("lightblue"))
shape.draw(win)
```




```
from Graphics import *
shape = Picture("http://4.bp.blogspot.com/__Y5vVtebEEE/SIeNV78jvAI/AAAAAAAABBY/9FIVWXy_Kho/
win = Window("Shapes", shape.width, shape.height)
shape.draw(win)
```


## Pixel

Pixels are generally not created outside of the context of a Picture.

Properties of pixels:

- x - (readonly) x location in picture from whence it came
- y - (readonly) y location in picture from whence it came
- picture - (readonly) reference to the picture from whence it came

Global Pixel Functions:

- getPixel(picture, x, y)
- getPixels(picture)
- setPixels(picture1, picture2)
- setPixel(picture, x, y, pixel)
- setPixel(picture, x, y, color)

Pixels don't have color themselves, but you can get the color components:

- getRed()
- getGreen()
- getBlue()
- getAlpha()
- getRGB()
- getRGBA()
- getColor()
- setColor(pixel, color)

- setRed(pixel, integer)
- setGreen(pixel, integer)
- setBlue(pixel, integer)
- setAlpha(pixel, integer)

What is the difference between Color and Pixel?

- Colors are independent
- Colors know their red, green, blue, and alpha
- Pixels depend on a particular Picture
- Pixels known their x, y, and associated picture

You can set the color of a pixel in a picture with:

- setPixel(pic, x, y, pixel)
- setPixel(pic, x, y, color)

### Colors

Colors are composed of 4 components: red, green, blue, and alpha. All values are integers between 0 (dark) and 255 (bright). Alpha is the transparency of the color. An alpha of 255 is completely opaque, and an alpha of 0 is completely transparent.

Constructors:

- Color(r, g, b)
- Color(color_name)
- Color(hexcolor)

Functions:

- makeColor(r, g, b)
- makeColor(color_name)
- makeColor(hexcolor)

Properties:

- red - (read/write) integer value of red component (0 - 255)
- green - (read/write) integer value of green component (0 - 255)
- blue - (read/write) integer value of blue component (0 - 255)
- alpha - (read/write) integer value of alpha component (0 - 255)

Color is used:

- for each Pixel of a Picture
- for the color of a Picture
- fill of any shape
- outline of any shape

See also the function pickAColor() from the Myro library.

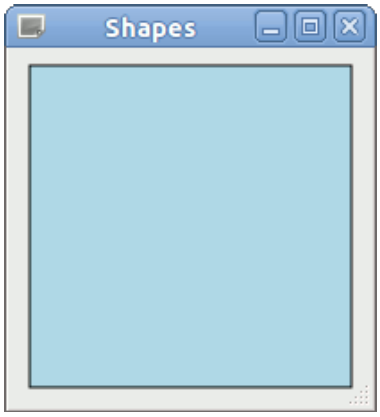
## Rectangle

Draw a rectangle or square.

- Rectangle((x1, y1), (x2, y2))
- Rectangle(Point(x1, y1), Point(x2, y2))

```
from Graphics import *
win = Window("Shapes", 200, 200)
```

```
shape = Rectangle((10, 10), (190, 190))
shape.fill = Color("lightblue")
shape.draw(win)
```

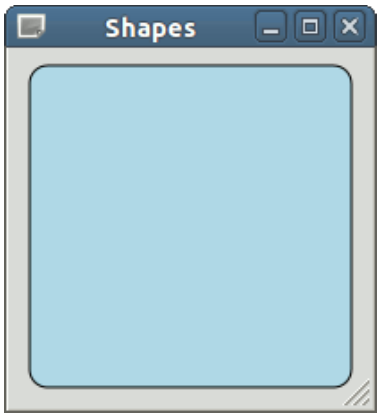



## RoundedRectangle

Draw a rounded rectangle or square. *New in Calico 1.0.4.*

- RoundedRectangle((x1, y1), (x2, y2), radius)
- RoundedRectangle(Point(x1, y1), Point(x2, y2), radius)

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = RoundedRectangle((10, 10), (190, 190), 10)
shape.fill = Color("lightblue")
shape.draw(win)
```

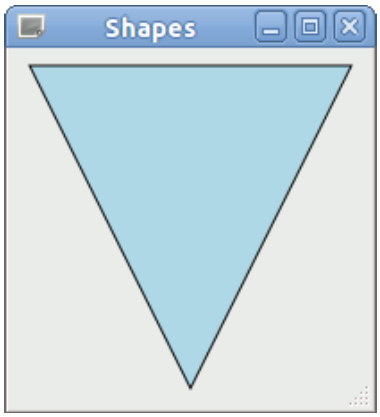



## Polygon

Draw a polygon by listing its points.

- Polygon((x1, y1), ...)
- Polygon(Point(x1, y1), ...)

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = Polygon((10, 10), (190, 10), (100, 190))
shape.fill = Color("lightblue")
shape.draw(win)
```

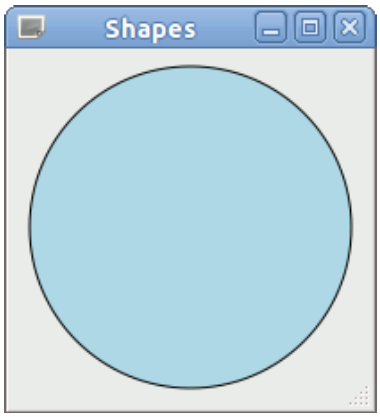


# Circle

Draw a circle.

- Circle((x, y), radius)
- Circle(Point(x, y), radius)

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = Circle((100, 100), 90)
shape.fill = Color("lightblue")
shape.draw(win)
```




# Oval

Draw an oval.

- Oval((x, y), xradius, yradius)
- Oval(Point(x, y), xradius, yradius)

xradius is the distance (the width) when initially drawn; yradius is the distance (the height) when initially drawn.

```
# Pulse Example
# Indicator that the computer is busy
# Doug Blank <dblank@cs.brynmawr.edu>

from Graphics import *
from Myro import wait
import math

win = Window()

alphas = list(reversed([x/10 * 255 for x in range(10)]))

ovals = []
for i in range(0, 360, 36):
    oval = Oval((150, 150), 50, 20)
    oval.rotate(-i)
    oval.color = Color("purple")
    position = int(abs(oval.rotation/(2 * math.pi) * 10))
    oval.color.alpha = alphas[9 - position]
    oval.draw(win)
    oval.forward(60)
    ovals.append(oval)

alphas.append(alphas.pop(0))

while True:
    for oval in ovals:
        position = int(abs(oval.rotation/(2 * math.pi) * 10))
        oval.color.alpha = alphas[9 - position]
        win.step(.0075)
    alphas.append(alphas.pop(0))
```

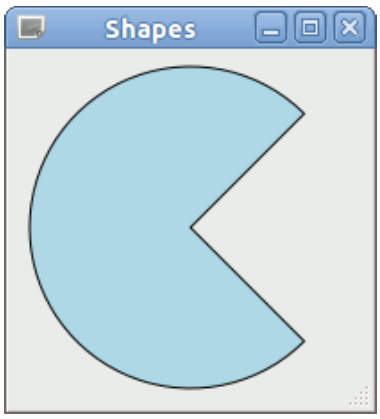
# Pie

Draw a slice of pie.

- Pie((x, y), radius, startDegree, stopDegree) - zero to right
- Pie(Point(x, y), radius, startDegree, stopDegree) - zero to right

The following example draws a Pacman-like shape.

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = Pie((100, 100), 90, 45, 360 - 45)
shape.fill = Color("lightblue")
shape.draw(win)
```

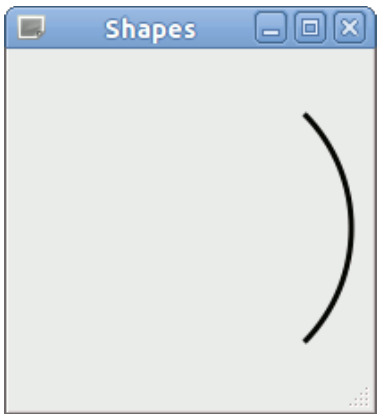



# Arc

Draw an arc.

- Arc((x, y), radius, startDegree, stopDegree) - zero to right
- Arc(Point(x, y), radius, startDegree, stopDegree) - zero to right

```
from Graphics import *
win = Window("Shapes", 200, 200)
shape = Arc((100, 100), 90, 360 - 45, 360 + 45)
shape.border = 3
shape.draw(win)
```




# GUI Widgets

Calico Graphics also has a set of GUI widgets. These have additional uses for providing control.

### Button

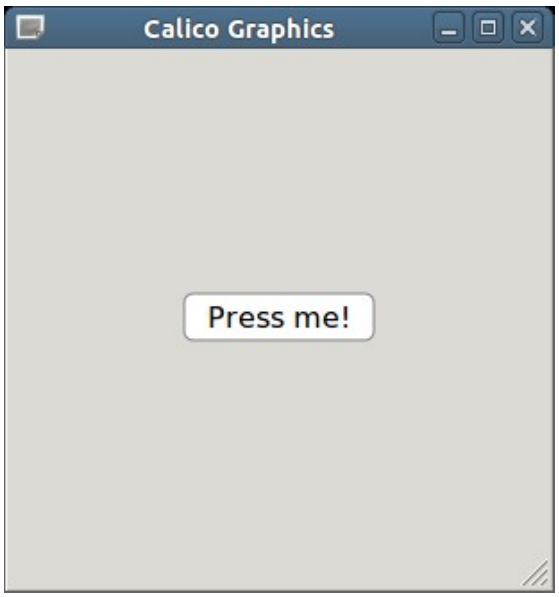
Draw an button, and alternatively connect a function to it. *New in Calico 1.0.4; updated in 1.1.0*

- Button((x, y), text)
- Button(Point(x, y), text)

You connect a function to it like so:

```
button.connect("click", function)
```

```
from Graphics import *
win = Window()
button = Button(Point(150, 150), "Press me!")
button.draw(win)

def printit(o, e):
    print(e)

button.connect("click", printit)
```

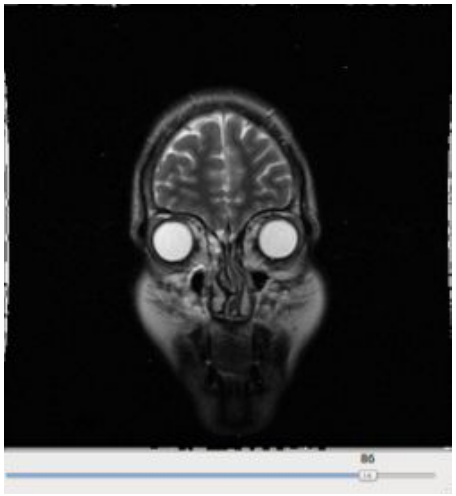



### HSlider

Horizontal slider. *New in Calico Graphics 1.1.0.*

- HSlider((x,y), width)
- HSlider(Point(x,y), width)

```
from Graphics import *
from Myro import getFilenames

files = getFilenames("../images/brain/*.jpg")

pics = []

for file in files:
    pics.append(makePicture(file))

window = Window("Laura's Brain", pics[0].width, pics[1].height + 50)

last = 0
pics[last].draw(window)
pics[last].moveTo(pics[0].width/2, pics[0].height/2)

slider = HSlider((0,pics[0].height), pics[0].width)
slider.draw(window)

def showimage(obj, event):
    global last
    v = event.value
    pos = int(v/101 * len(pics) )
    if pos != last:
        window.undraw(pics[last])
        pics[pos].draw(window)
        pics[pos].moveTo(pics[0].width/2, pics[0].height/2)
        last = pos

slider.connect("change-value", showimage)
```



## Groups of Shapes

There are two ways to group shapes: the Group and the Frame.

The Group is used for making a group out of existing shapes, so that you can easily perform a rotation to all of them.

The Frame is useful for creating a set of shapes all in the same frame of reference. Actually, a Frame is just an invisible Shape, and behaves like all shapes. You create a Frame, and then draw objects onto it.

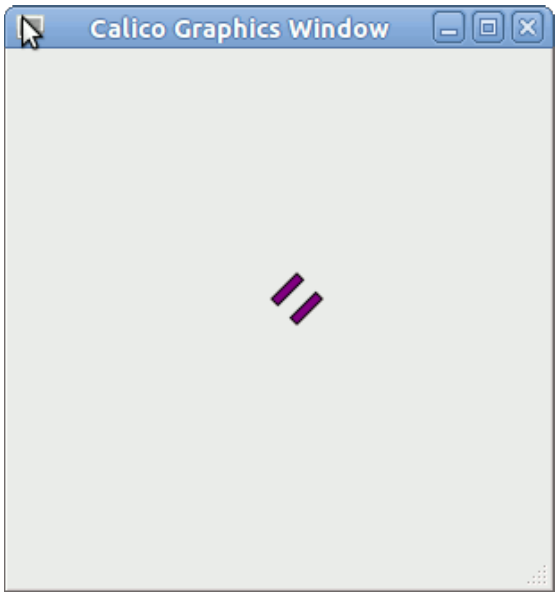
## Frame

- Frame(x, y)
- Frame((x, y))

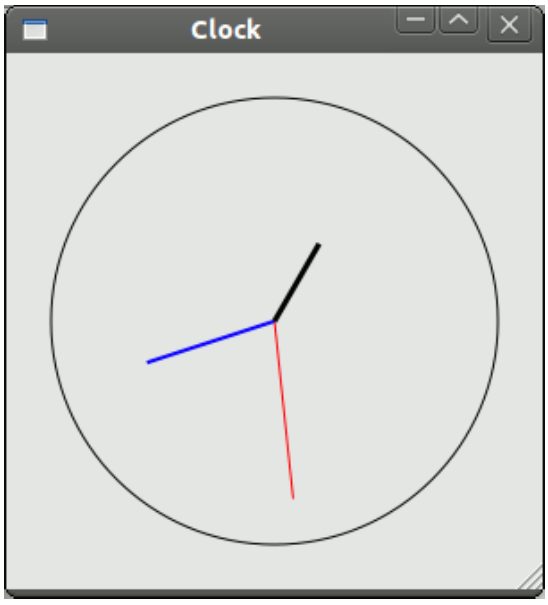
You create a frame, and then draw onto it:

- frame = frame(150, 150)
- rectangle.draw(frame)

```
from Graphics import *
win = Window()
car = Frame(150, 150)
wheel1 = Rectangle((-10, -10), (10, -5))
wheel2 = Rectangle((-10, 10), (10, 5))
wheel1.draw(car)
wheel2.draw(car)
car.draw(win)

car.rotate(45)
car.forward(10)
```



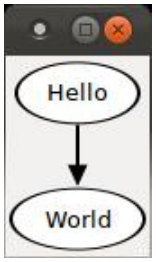

```
from Graphics import *
import time

win = Window("Clock")
win.mode = "manual"

face = Circle((150, 150), 125)
face.fill = None
face.draw(win)

s = Frame(150, 150)
line = Line((0, 0), (100, 0))
line.color = Color("red")
line.draw(s)

m = Frame(150, 150)
line = Line((0, 0), (75, 0))
line.color = Color("blue")
line.border = 2
line.draw(m)

h = Frame(150, 150)
line = Line((0, 0), (50, 0))
line.color = Color("black")
line.border = 3
line.draw(h)

s.draw(win)
```



```
m.draw(win)
h.draw(win)

def main():
    while True:
        t = time.localtime()
        s.rotateTo(t[5]/60 * 360 - 90)
        m.rotateTo(t[4]/60 * 360 - 90)
        h.rotateTo(t[3]/12 * 360 - 90)
        win.step(1)

win.run(main)
```

## Graphs

```
from Graphics import *
g = Graph()
g.addEdge("Hello", "World")
g.layout()
g.draw()
```




Graph properties:

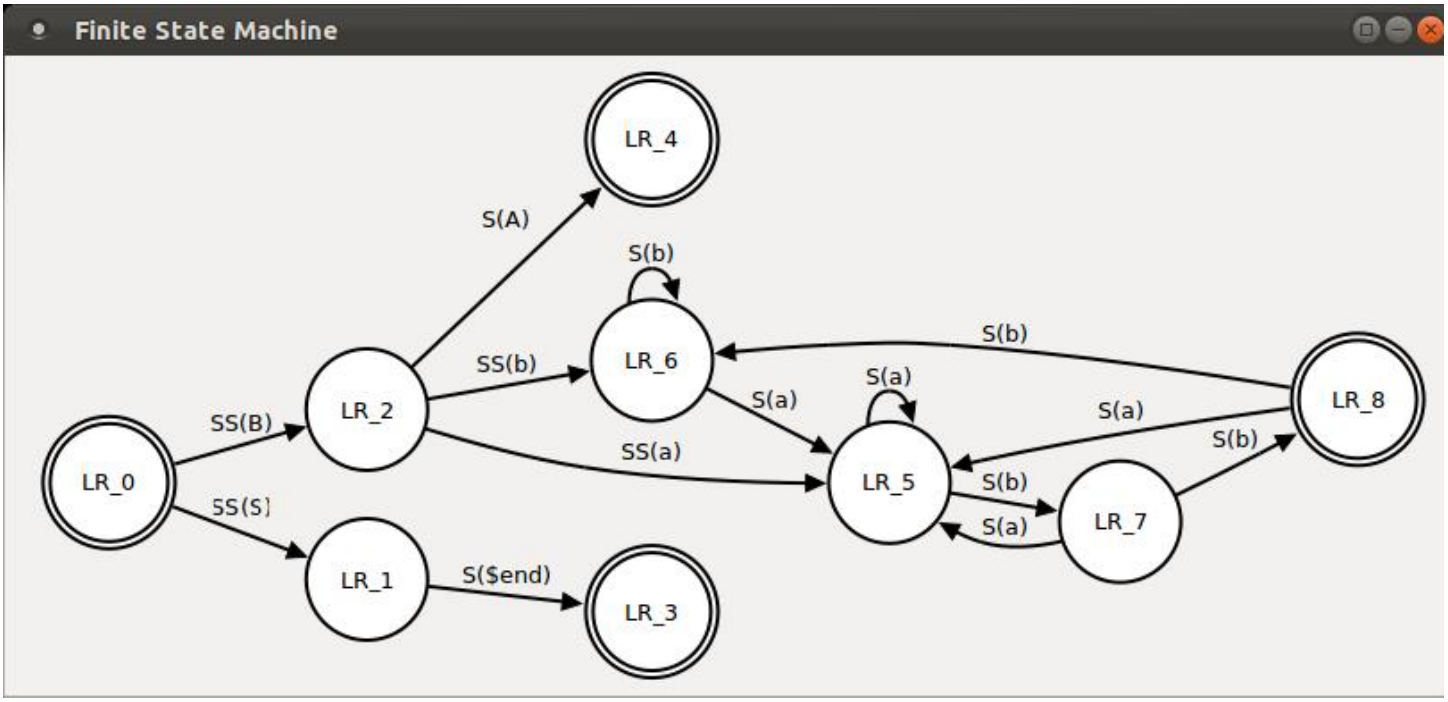
- addEdge(from, to)
- addNode(name)
- draw(window | None)
- edges - the edges
- getEdgeLines()
- getNode()
- graph - the graph
- graph_count - count
- graphEdges - the edges
- graphNodes - the nodes
- layout() - call GraphViz to get layout
- layout(list) - call GraphViz to get Binary Tree layout of a list, where list is [left-tree, node, right-tree]
- layout(list, a, b, c) - call GraphViz to get Binary Tree layout of a list, where a is left-position, b is node, and c is right-position
- lookupNode()
- options - the options
- parser - the parser
- post_text - the text after GraphViz processing
- pre_text - the text before GraphViz processing
- processDot() - call the dot external
- recurseEdges() - internal call to process edges
- recurseNodes() - internal call to process nodes
- vertices - intenal vertices
- window - the Graphics window, if one

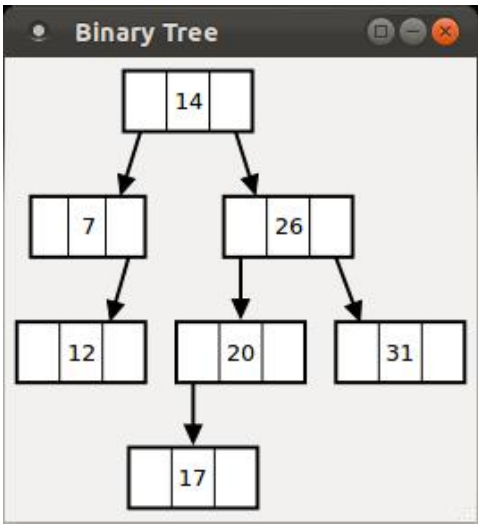
### Scheme Example

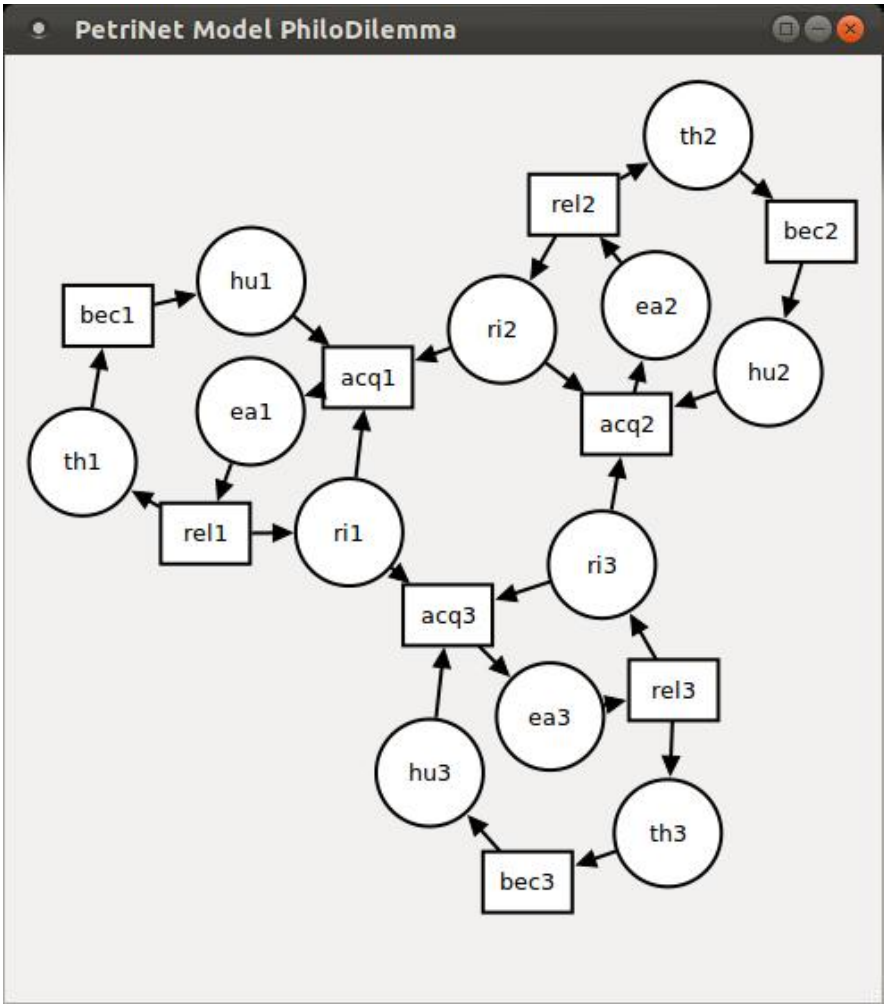
```
(define show-tree
   (lambda (tree)
      (using "Graphics")
      (define! g (Graphics.Graph))
      (g.layout tree 1 0 2) ;; left-index root-index right-index
```

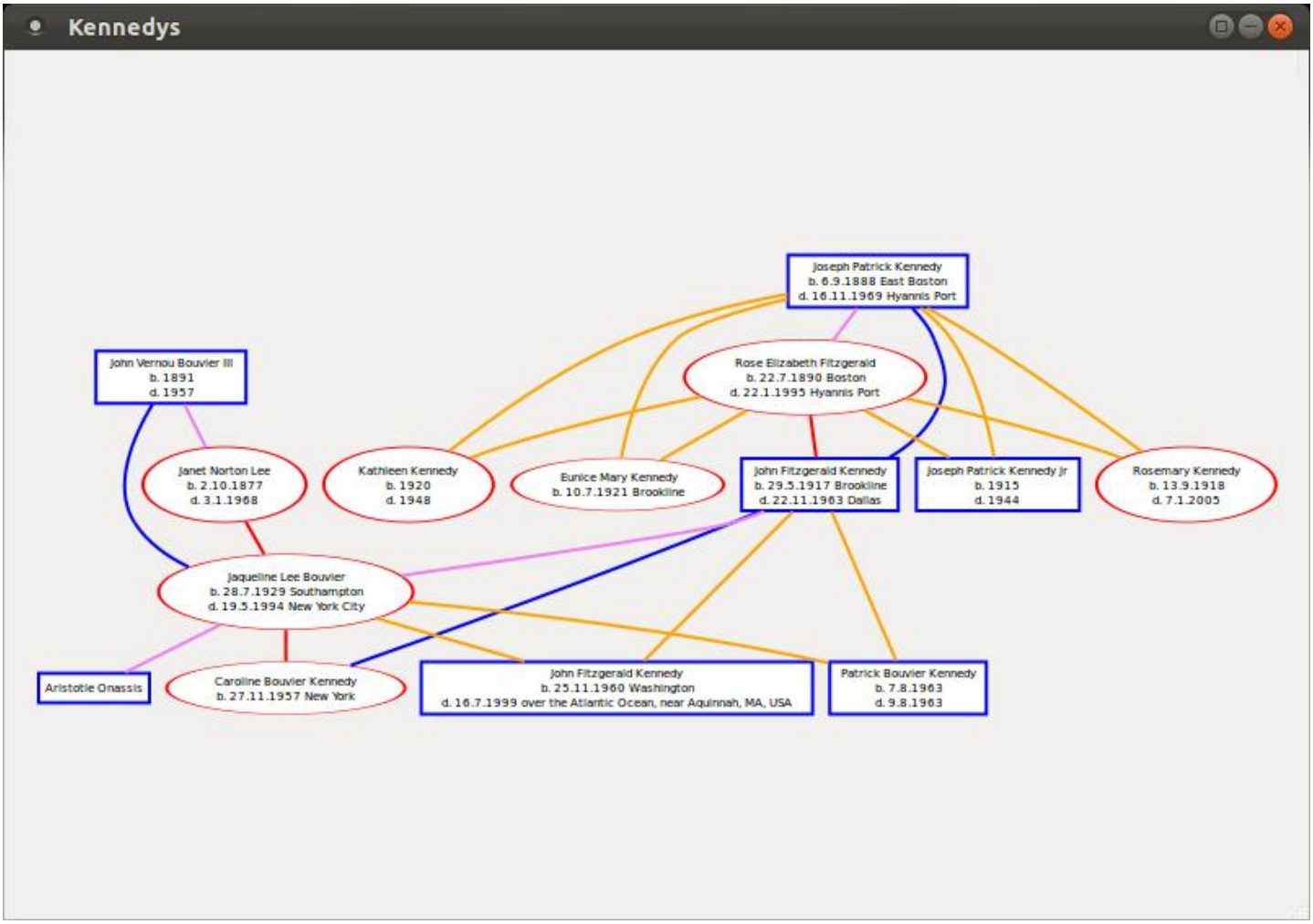
```
      (g.draw)))

(show-tree '(42 () ()))
```

Examples:




PetriNet Model PhiloDilemma
th2
rel2
bec2
hu1
bec1
ri2
ea2
hu2
acq1
ea1
acq2
th1
rel1
ri1
ri3
acq3
ea3
rel3
hu3
th3
bec3




## Group

Create a group of shapes.

- Group(shape1, shape2, ...)

By creating a group, you can move/change all of the shapes at once.

```
from Graphics import *
win = Window("USFlag", 700, 400)

def make_star(x, y, segment):
    arrow = Arrow(Point(x, y), 1)
    arrow.draw(win)
    arrow.penDown()
    for i in range(5):
        arrow.forward(segment)
        arrow.rotate(72)
        arrow.forward(segment)
        arrow.rotate(-72)
        arrow.rotate(-72)
    polygon = Polygon(*arrow.penUp())
    polygon.draw(win)
    polygon.color = makeColor("white")
    arrow.undraw()
    return polygon

for row in range(13):
    band = Rectangle(Point(0,row * 400/13), Point(700, row * 400/13 + 400/13))
    band.draw(win)
    if row % 2 == 1: # odd, white
        band.color = makeColor("white")
    else:
```

```
        band.color = makeColor("red")

blue = Rectangle(Point(0,0), Point(300, 214))
blue.color = makeColor("blue")
blue.draw(win)
stars = []
for col in range(6):
    for row in range(9):
        if row % 2 == 1: # odd row
            if col == 5:
                continue
            x = col * 50 + 25
        else:
            x = col * 50
        y = row * 22
        star = make_star(x + 10, y + 13, 5)
        stars.append(star)
```

And now to demonstrate the Group() constructor:

```
def animate():
    g = Group(*stars)
    win.mode = "manual"
    for i in range(20):
        g.rotate(10)
        win.step(.5)
```

# Window Features

Windows can operate in a number of modes.

## modes

- "auto"
- "manual"
- "physics"
- "bitmap"

You can also "run" the window, which automatically calls the window.step() function, or you can call your own function.

- win.run()
- win.run(function)

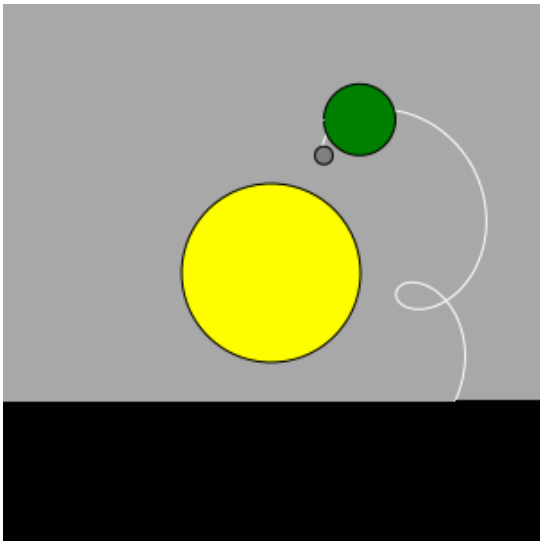
## Miscellaneous

Shapes will be stacked in the order that they were drawn on the window (eg, first drawn are on the bottom). However, one can change the order with the following methods:

- window.clear() - clears all shapes drawn on window
- window.stackOnTop(shape) - moves shape to top
- window.stackOnBottom(shape) - moves shape to bottom

```
from Graphics import *

win = Window()
win.setBackground(Color("black"))

sun = Circle((150, 150), 50)
sun.fill = Color("yellow")
sun.draw(win)

earth = Circle((70, 70), 20)
```

```
earth.fill = Color("green")
earth.draw(sun)

moon = Circle((20, 20), 5)
moon.fill = Color("grey")
moon.draw(earth)

pen = Pen(Color("white"), True) # True means that pen is down
pen.draw(win)
pen.stackOnBottom()

def main():
    win.mode = "manual"
    for s in range(360):
        sun.rotate(1)
        earth.rotate(5)
        win.step(.1)
        pen.appendPath(Point(moon.gx, moon.gy)) # gx,gy is global position

win.run(main)
```



## Physics

The window can be in "physics" mode. The following window properties are then useful.

- win.gravity = Vector(x, y)
- win.time - total time of simulation so far
- win.simulationStepTime - time to advance simulation on each step (default is .01 seconds)

The following shape properties are then useful in "physics" mode.

- shape.body.ApplyForce(Vector(x, y))
- shape.body.ResetDynamics()
- shape.wrap = boolean
- shape.bounce - 1.0 is 100% bounce; 0.0 is no bounce
- shape.friction - amount of friction

Demonstrations:

- http://www.youtube.com/watch?v=L3P5t8hAhe4 - Word Physics
- http://www.youtube.com/watch?v=Rl0tVqsf71g - Word Physics II
- http://www.youtube.com/watch?v=yLZJmyQEALA - Angry Blocks
- http://www.youtube.com/watch?v=XDHLbi1pl9U - In slow motion

# Low Level Graphics

You can also create your own shapes in Calico Graphics. One easy way is to subclass the **Shape** class and override the **render** method, like so:

```
class MyShape(Shape):
    def render(self, cr):
        cr.LineTo(0, 0)
        cr.LineTo(100, 100)
        cr.Stroke()
```

Render takes a Cairo Graphics context (cr). You can read more about what you can do with Cairo Graphics (http://www.mono-project.com/Mono.Cairo_Tutorial) . Those examples are in C#.

To convert to Calico Python:

- leave out the word "new"

- Color is the Calico Graphics Color so use Color(...).getCairo()
- semi-colons are optional
- don't list the type of variables

For example, this C# code:

```
cr.LineWidth = 0.1;
cr.Color = new Color(0, 0, 0);
cr.Rectangle(0.25, 0.25, 0.5, 0.5);
cr.Stroke();
```

would become this Calico Python code:

```
cr.LineWidth = 0.1
cr.Color = Color(0, 0, 0).getCairo()
cr.Rectangle(0.25, 0.25, 0.5, 0.5)
cr.Stroke()
```

Full example:

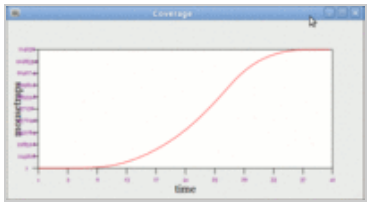
```
from Graphics import *
win = Window()
class MyShape(Shape):
    def render(self, cr):
        cr.LineTo(0, 0)
        cr.LineTo(100, 100)
        cr.Stroke()
s = MyShape()
s.draw(win)
```

# Plot

The Plot object allows you to create a graph of data.

- Plot("Title", width, height)
- Plot(data)
- plot.xLabel.text
- plot.yLabel.text
- plot.append(data)

```
from Graphics import Plot

plot = Plot("Sample Plot", 600, 300)
plot.xLabel.text = "time"
plot.yLabel.text = "balls in the air"

for data in [10, 30, 40, 50, 0, 100, 110, 40, 50]:
    plot.append(data)
```



Sample Plot